tsi Documentation

Release 0.1

Phil Maker

January 19, 2016

CONTENTS

Contents:

CHAPTER

ONE

INTRODUCTION

This is a python setup for analysing irregularly sampled time series data using pandas, numpy, etc.

CHAPTER

TWO

OVERVIEW

2.1 Input Data Format

The incoming data is kept in ASIM format using ISO8601 time stamps and looks like:

```
t,R_K_PG_BULM_Fed3Pact
1999-12-31 14:30:00.000+00:00, NaN
2015-06-18 16:06:54.000+00:00, NaN
2015-06-18 16:14:00.000+00:00, 20.00000
2015-06-18 16:14:10.000+00:00, NaN
2015-07-18 16:14:14.535+00:00, 50.00000
2015-07-18 16:14:30.535+00:00, NaN
```

The timestamps are all in UCT and

The values are plain old numbers using NaN to indicate loss of signal, e.g. a communications link failure, Values maintain their old value until the next sample.

2.2 Reading data into a Pandas DataFrame

Multiple data files can be read into a single pandas DataFrame, e.g reading Test1.csv and Test2.csv containing the data for all timestamps in the series in a DataFrame. Extra fields include:

1. t - the timestamp for a particular point in time which is the index.
2. Test1, Test2 - the actual data (in this example).
3. dt - the duration that all variables remain the same, i.e. the time in seconds to the next row.
4. wTest1, wTest2 - the time intergral for Test1 at this time.
5. tTest1, tTest2 - the duration for each value or 0 if it is not a number (NaN)

The frame data looks like:

```
*                            Test1   Test2   Remarks            dt
t
```

```
1999-12-31 14:30:00.000000    NaN    NaN      NaN  0.000000e+00
1999-12-31 14:30:00.000000    NaN    NaN      NaN  4.879930e+08
2015-06-18 16:06:54.000000    NaN    NaN      NaN  0.000000e+00
2015-06-18 16:06:54.000000    NaN    NaN      NaN  4.260000e+02
2015-06-18 16:14:00.000000     20    NaN      NaN  2.000000e+00
2015-06-18 16:14:02.000000     20    120      NaN  8.000000e+00
2015-06-18 16:14:10.000000    NaN    120      NaN  2.009999e+00
2015-06-18 16:14:12.009999    NaN    NaN      NaN  2.592003e+06
2015-07-18 16:14:14.535000     50    NaN      NaN  2.000000e+00
2015-07-18 16:14:16.535000     50      5      NaN  1.400000e+01
2015-07-18 16:14:30.535000    NaN      5      NaN  0.000000e+00
2015-07-18 16:14:30.535000    NaN    NaN      NaN           NaN
```

Along with w*, t* values which will look like:

```
*                          wTest1  tTest1     wTest2     tTest2
t
1999-12-31 14:30:00.000000    NaN       0        NaN   0.000000
1999-12-31 14:30:00.000000    NaN       0        NaN   0.000000
2015-06-18 16:06:54.000000    NaN       0        NaN   0.000000
2015-06-18 16:06:54.000000    NaN       0        NaN   0.000000
2015-06-18 16:14:00.000000     40       2        NaN   0.000000
2015-06-18 16:14:02.000000    160       8  960.00000   8.000000
2015-06-18 16:14:10.000000    NaN       0  241.19988   2.009999
2015-06-18 16:14:12.009999    NaN       0        NaN   0.000000
2015-07-18 16:14:14.535000    100       2        NaN   0.000000
2015-07-18 16:14:16.535000    700      14   70.00000  14.000000
2015-07-18 16:14:30.535000    NaN       0    0.00000   0.000000
2015-07-18 16:14:30.535000    NaN       0        NaN   0.000000
```

Where Test1.csv is:

```
t,R_K_PG_BULM_Fed3Pact
1999-12-31 14:30:00.000+00:00, NaN
2015-06-18 16:06:54.000+00:00, NaN
2015-06-18 16:14:00.000+00:00, 20.00000
2015-06-18 16:14:10.000+00:00, NaN
2015-07-18 16:14:14.535+00:00, 50.00000
2015-07-18 16:14:30.535+00:00, NaN
```

And Test2.csv is:

```
t,R_K_PG_BULM_PvPact
1999-12-31 14:30:00.000+00:00, NaN
2015-06-18 16:06:54.000+00:00, NaN
2015-06-18 16:14:02.000+00:00, 120.00000
2015-06-18 16:14:12.010+00:00, NaN
2015-07-18 16:14:16.535+00:00, 5.00000
2015-07-18 16:14:30.535+00:00, NaN
```

CHAPTER

THREE

SETUP, CONVENTIONS AND TOOLS

3.1 Conventions

1. Use Python version 3
2. Use the pandas, numpy,scipy and matplotlib
3. Documentation and tests will in Sphinx using the google documentation format rather numpydoc (which in turn is better than Sphinx .rst).

3.2 Setup

1. Install Sphinx
2. Configure the documentation in the doc directory using `spinx-quickstart` to configure it. Select the following options:
   (a) autodoc
   (b) doctest
   (c) document coverage
   (d) pngmath
   (e) viewcode
3. Install `numpydoc` and add into the extensions list in ‘‘conf.py’‘

   Inline literal start-string without end-string.
4. Install `napoleon` extension and enable it
5. Add a reference to the various `intro.rst` and `history.rst` into `index.rst`
6. Add google doc to the modules, e.g. `exdoc.py`.
7. See ‘‘‘exdoc.py” for examples following the google conventions.

   Inline literal start-string without end-string.

   Inline interpreted text or phrase reference start-string without end-string.

CHAPTER

FOUR

PYTHON-TSI-TOOLS

4.1 exdoc module

Example Google style docstrings.

This module demonstrates documentation as specified by the Google Python Style Guide. Docstrings may extend over multiple lines. Sections are created with a section header and a colon followed by a block of indented text.

Example

Examples can be given using either the `Example` or `Examples` sections. Sections support any reStructuredText formatting, including literal blocks:

```
$ python example_google.py
```

Section breaks are created by resuming unindented text. Section breaks are also implicitly created anytime a new section starts.

`exdoc.`**`module_level_variable1`**
int

Module level variables may be documented in either the `Attributes` section of the module docstring, or in an inline docstring immediately following the variable.

Either form is acceptable, but the two should not be mixed. Choose one convention to document module level variables and be consistent with it.

class `exdoc.`**`ExampleClass`**(*param1*, *param2*, *param3*)
Bases: `object`

The summary line for a class docstring should fit on one line.

If the class has public attributes, they may be documented here in an `Attributes` section and follow the same formatting as a function's `Args` section. Alternatively, attributes may be documented inline with the attribute's declaration (see __init__ method below).

Properties created with the `@property` decorator should be documented in the property's getter method.

Attribute and property types – if given – should be specified according to PEP 484, though PEP 484 conformance isn't required or enforced.

attr1
str

Description of *attr1*.

attr2
Optional[int]

Description of *attr2*.

__special__()
By default special members with docstrings are included.

Special members are any methods or attributes that start with and end with a double underscore. Any special member with a docstring will be included in the output.

This behavior can be disabled by changing the following setting in Sphinx's conf.py:

```
napoleon_include_special_with_doc = False
```

attr3 = None
Doc comment *inline* with attribute

attr4 = None
List[str]: Doc comment *before* attribute, with type specified

attr5 = None
Optional[str]: Docstring *after* attribute, with type specified.

example_method(*param1*, *param2*)
Class methods are similar to regular functions.

Note: Do not include the *self* parameter in the `Args` section.

Parameters

- **param1** – The first parameter.
- **param2** – The second parameter.

Returns True if successful, False otherwise.

readonly_property
str: Properties should be documented in their getter method.

readwrite_property
List[str]: Properties with both a getter and setter should only be documented in their getter method.

If the setter method contains notable behavior, it should be mentioned here.

exception exdoc.**ExampleError**(*msg*, *code*)

Bases: Exception

Exceptions are documented in the same way as classes.

The __init__ method may be documented in either the class level docstring, or as a docstring on the __init__ method itself.

Either form is acceptable, but the two should not be mixed. Choose one convention to document the __init__ method and be consistent with it.

Note: Do not include the *self* parameter in the Args section.

Parameters

- **msg** (*str*) – Human readable string describing the exception.
- **code** (*Optional[int]*) – Error code.

msg

str

Human readable string describing the exception.

code

int

Exception error code.

exdoc.**example_generator**(*n*)

Generators have a Yields section instead of a Returns section.

Parameters **n** (*int*) – The upper limit of the range to generate, from 0 to *n* - 1.

Yields *int* – The next number in the range of 0 to *n* - 1.

Examples

Examples should be written in doctest format, and should illustrate how to use the function.

```
>>> print([0,1,2,3])
[0, 1, 2, 3]
```

exdoc.**main**()

The main program

exdoc.**module_level_function**(*param1*, *param2=None*, **args*, ***kwargs*)

This is an example of a module level function.

Function parameters should be documented in the `Args` section. The name of each parameter is required. The type and description of each parameter is optional, but should be included if not obvious.

Parameter types – if given – should be specified according to PEP 484, though PEP 484 conformance isn't required or enforced.

If *args or **kwargs are accepted, they should be listed as `*args` and `**kwargs`.

The format for a parameter is:

```
name (type): description
    The description may span multiple lines. Following
    lines should be indented. The "(type)" is optional.

    Multiple paragraphs are supported in parameter
    descriptions.
```

Parameters

- **param1** (*int*) – The first parameter.
- **param2** (*Optional[str]*) – The second parameter. Defaults to None. Second line of description should be indented.
- ***args** – Variable length argument list.
- ****kwargs** – Arbitrary keyword arguments.

Returns

True if successful, False otherwise.

The return type is optional and may be specified at the beginning of the `Returns` section followed by a colon.

The `Returns` section may span multiple lines and paragraphs. Following lines should be indented to match the first line.

The `Returns` section supports any reStructuredText formatting, including literal blocks:

```
{
    'param1': param1,
    'param2': param2
}
```

Return type bool

Raises

- `AttributeError` – The `Raises` section is a list of all exceptions that are relevant to the interface.
- `ValueError` – If *param2* is equal to *param1*.

exdoc.**module_level_variable2 = 98765**
int: Module level variable documented inline.

The docstring may span multiple lines. The type may optionally be specified on the first line, separated by a colon.

4.2 tsi module

tsi

This module....

Example

Example 1 with literal block:: $ echo hello

TODO:

1. Break this apart into individual components

tsi.**fntovar**(*fn*)
Convert filename to variable name

tsi.**getdf**(*pats*)
Return DataFrame from files matching members of pats.

pats - list of glob style pattern matching the files to process.

tsi.**hists = {}**
{var->[(when,what)]}: history of var as a list of (when,what) events

This is used as the basic representation for history of variables before we mangle it into the various pandas DataFrame representations.

tsi.**limit**(*v*, *low*, *high*)
limit v between low and high

tsi.**main**()
Do the work

tsi.**makedt**(*df*)
Return a dataframe with new dt, w* and t* Series.

The new dataframe contains:

df['dt'] - the difference in time between this samples. df['w' + var] - the time weighted value for var. df['t' + var] - the dt for var if it is not nan otherwise 0

tsi.**nan = nan**
float: just nan for us to use

tsi.**offset**(*v*, *o=0*)
offset v by o

tsi.**options = {'-trace': False, '-test': 1, '-profile_main': False, '-show_options': False}**
dictionary of command line options

tsi.**plotPdf**(*fn*, ***kwopts*)

tsi.**profile**(*c*)
profile code c Args: c (str): command to profile

tsi.**rest**()
Just a block to keep scrap code in

tsi.**rest2**()

tsi.**scale**(*v*, *p=1*)
scale v by p

tsi.**showeval**(*s*)
print a string followed by its evaluation

Note that python does not have an uplevel(3tcl) command like TCL and so need to use globals or the locals=dict argument. There must be a better way.

tsi.**tdsecs**(*td*)
convert a timedelta to a number in seconds

Parameters **ts** (*timedelta*) – a timedelta from pandas/numpy

Returns representation in seconds of delta

Return type float

Examples

None

tsi.**test1**()
run a simple test

Note

tsi.**test1_2**()
run a simple test

Note

tsi.**tformat**(*s*)
convert a float of seconds since epoch to an ISO8601 date

There is still a bit of representation error in this component. Why we aren't using numpy datetime64.

Parameters **s** (*float*) – seconds since UNIX epoch.

Returns formatted time

Return type str

Examples

```
>>> import tsi
>>> print(tsi.tformat(1000000000.0))
2001-09-09T01:46:40+00:00
```

```
>>> import tsi
>>> print(tsi.tformat(1000000000.988))
2001-09-09T01:46:40.988000+00:00
```

Note that we get representation errors in the following two tests.

!>>> import tsi !>>> print(tsi.tformat(1000000000.989)) 2001-09-09T01:46:40.989000+00:00

!>>> import tsi !>>> print(tsi.tformat(tsi.tparse('2001-09-09T01:46:40.989000+00:00'))) 2001-09-09T01:46:40.989000+00:00

tsi.**tparse**(*t*)

convert a ISO8601 timestamp to a float of seconds since UNIX epoch.

Parameters **t** (*str*) – ISO8601 timestamp

Returns if t is correct, otherwise fails

Return type timeStamp

Examples

```
>>> import tsi
>>> print(tsi.tparse('2001-09-09T01:46:40+00:00'))
1000000000.0
```

```
>>> import tsi
>>> print(tsi.tparse('2001-09-09T01:46:40Z'))
1000000000.0
```

```
>>> import tsi
>>> print(tsi.tparse('20010909T014640Z'))
1000000000.0
```

```
>>> import tsi
>>> print(tsi.tparse('2001-09-09T01:46:40.123+00:00'))
1000000000.123
```

```
>>> import tsi
>>> print(tsi.tparse('2001-09-09 01:46:40.123+00:00'))
1000000000.123
```

```
>>> import tsi
>>> print(tsi.tparse('2001-09-09 01:46:40.978612+0000'))
1000000000.978612
```

```
>>> import tsi
>>> print(tsi.tparse('2099-09-09 01:46:40.978612+0000'))
4092601600.978612
```

tsi.**ts2csv**(*fd*)
print ts stat to fd

tsi.**ts2csvbody**(*fd*)
print the body

tsi.**ts2csvheader**(*fd*)
print the header line

tsi.**tsevents**()
converts hists[] to [(when,var,what)...]

Returns: [(when,var,what)]: similar to an alarm log

tsi.**tsmean**(*df*, *v*)
Return the time weighted average for v in DataFrame df

tsi.**tsread**(*fn*)
read file fn and convert [(when,what)...]

Parameters **fn** (*str*) – filename to read which is in ASIM format

Returns **[** – list of when, what events

Return type when,what

tsi.**tsreadfiles**(*pat*, *rename*)
Read all files matching glob pat and rename var using rename

Args:

pat (str): glob pattern for matching files rename (hook): function renaming variables from file to varname

Returns: nothing

tsi.**tsstates**()
converts hists to [when, {var->what}] by remembering state

For example [a:[(100,1),(200,10)],b:[(10,-1),(250,-11)]] converts to [(10,{a:nan,b:-1}),(100,{a:1,b:-1}),(200,{a:10,b:-1}),

Unexpected indentation.

(250,{a:10,b:-11})]

Examples

None yet till I redo it

Returns: [when, {var->what}]:

tsi.**tssummary**(*df*, *v*)
Return a summary of the variable v in dataframe df.

tsi.**tsvars**()
returns variables we have history for

Returns: [str]: sorted list of variable names

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

e

t

Symbols

A

C

E

F

G

H

L

M

N

O

P

R

S

T